Energy-Efficient Memory Mappings based on Partial WCET Analysis and Multi-Retention Time STT-RAM

Rabab Bouziane, Erven Rohou, Abdoulaye Gamatié

Energy-Efficient Memory Mappings based on Partial WCET Analysis and Multi-Retention Time STT-RAM

Rabab Bouziane
Univ Rennes, Inria, CNRS, IRISA
rabab.bouziane@inria.fr

Erven Rohou
Univ Rennes, Inria, CNRS, IRISA
erven.rohou@inria.fr

Abdoulaye Gamatié
CNRS, Univ Montpellier, LIRMM
abdoulaye.gamatie@lirmm.fr

ABSTRACT

Energy-efficiency has become one major challenge in both embedded and high-performance computing. Different approaches have been investigated to solve the challenge, e.g., heterogeneous multicore, system runtime and device-level power management. This paper targets emerging non volatile memories (NVMs), through Spin-Transfer Torque RAM (STT-RAM), which inherently have quasi-null leakage. This enables to reduce the static power consumption, which tends to become dominant in modern systems. The usage of NVM in memory hierarchy comes however at the cost of expensive write operations in terms of latency and energy. In order to mitigate this detrimental feature, this paper leverages the notion of *delta worst-case execution time* (δ-WCET), which consists of partial WCET estimates. From program analysis, δ-WCETs are determined and used to safely allocate data to NVM memory banks with variable data retention times. The δ-WCET analysis computes the WCET between any two locations in a function code, i.e., between basic blocks or instructions. Our approach is validated on the Mälardalen benchmark suite and significant memory dynamic energy reductions (up to 80 %, and 66 % on average) are reported.

ACM Reference Format:
Rabab Bouziane, Erven Rohou, and Abdoulaye Gamatié. 2018. Energy-Efficient Memory Mappings based on Partial WCET Analysis and Multi-Retention Time STT-RAM. In *26th International Conference on Real-Time Networks and Systems (RTNS '18), October 10–12, 2018, Chasseneuil-du-Poitou, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3273905.3273908

RTNS '18, October 10–12, 2018, Chasseneuil-du-Poitou, France

ACM ISBN 978-1-4503-6463-8/18/10…$15.00
https://doi.org/10.1145/3273905.3273908

1 INTRODUCTION

Energy-efficiency has become one major challenge in several computing domains, including embedded and high-performance computing domains. Examples of approaches for addressing the issue are system runtime management, heterogeneous multicores and device-level power management.

In the particular case of embedded domain, e.g., the Internet of Things (IoT), devices are increasingly considering non-volatile memories (NVMs) for their ability to favor normally-off computing that aggressively powers off devices. Thanks to their inherent non-volatility property, the information stored within the NVM can be safely recovered when the system is powered on later. During the switch-off period of the system, neither leakage nor refresh power are involved to preserve the information. This property of NVMs makes them highly attractive w.r.t. classical memory technologies such as SRAM or DRAM.

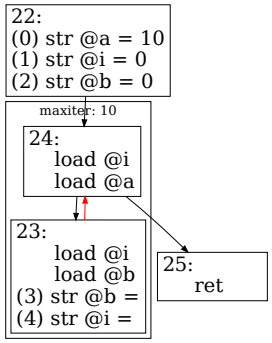
In the current study, we consider a specific emerging NVM, named Spin-Torque Transfer random access memory (STT-RAM) [2]. While such a technology has a quasi-null leakage, one major challenge concerns the mitigation of its high write latency and energy cost. Indeed, depending to the NVM technology, writes are several times more costly compared to SRAM or DRAM.

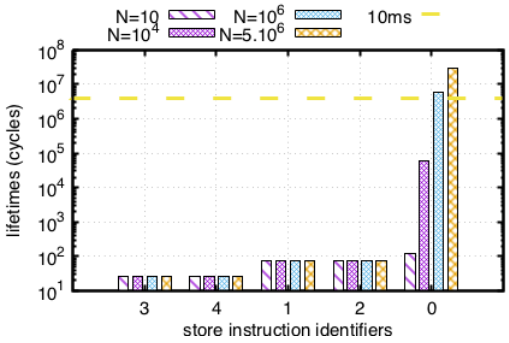
Non-volatility refers to extremely long retention time. Current non volatile RAMs (NVRAMs) are designed with 10+ years of data retention in order to favor higher memory reliability [16]. Yet, many other NVM technology designs are possible, which enable to build multi-bank memories with variable retention times [17]. This opens an interesting opportunity because memories with shorter retention times have much cheaper latency and energy costs, and typical variables in programs often have a much shorter lifetime. The next section motivates this opportunity.

1.1 Motivational Example

Consider the example of Figure 1. The instructions using the three variables a, b and i result in five different memory writes (store instructions) as shown in the control flow graph. The first three initialize the variables in block 22, the last two update the values of i (i++) and b (b=b+i) in block 23. The lifetime of a value is defined as the duration between its

```
#define N 10
int main(void)
{
  int a, b, i;
  a = N;
  b = 0;
  for (i=0; i<a; i++) {
    ANNOT_MAXITER(N);
    b=b+i;
  }
}
```





(a) motivational example

(b) control-flow graph and memory access instructions

(c) duration of the lifetimes created by each store instruction, for different numbers of iterations of the loop, and 10 ms threshold

Figure 1: A sample program (a), with its associated Control Flow Graph - CFG (b). The loop iterates ten times. ANNOT_MAXITER is a macro that stores this information in a dedicated ELF section of the binary, retrieved by the Heptane tool [7] and attached to the CFG. The lifetimes of its five store instructions are shown in (c).

Retention time	Read energy (nJ): α^R	Write energy (nJ): α^W
10 years	0.233	0.601
10 ms	0.233	0.269

Table 1: Two 512 KB NVM retention times [11]

definition (i.e., a write) and its last *use* (i.e., a load) before it is redefined. The lifetimes created by all the stores are very different. Store (0) in block 22 creates a lifetime that spans the entire duration of the program because it is defined at the beginning and read in each iteration of the loop. Conversely, the lifetimes defined by stores (1) and (2) in block 22 only reach the first iteration of block 23 in which they are redefined. The same holds for the data lifetimes defined at stores (3) and (4) in block 23, alive only across one iteration. Hence, even with a large value of N, most values written to memory only require a short retention time. This is shown in Figure 1 (c): only one value increases proportionally to N.

As an example, let us assume a flat memory (i.e., no cache), with the read/write energy costs from Khoshavi et al. [11], also summarized in Table 1. Let α_t^R (resp. α_t^W) be the cost a single read (resp. write) to a memory bank with retention time t, the cost of executing the program with a 10-years retention NVM memory is:

$$E_0 = N \times (4\alpha^R_{10yr} + 2\alpha^W_{10yr}) + 3\alpha^W_{10yr}$$

If a 10 ms retention memory bank is available, stores (1), (2), (3), and (4) can be assigned to it. For large values of N, store (0) must be assigned to the longer bank (for $N \geq 66655$, the lifetime exceed 400,002 cycles, i.e. 10 ms at 40 MHz as in our experimental setup). The energy cost therefore becomes:

$$E_1 = N \times (3\alpha^R_{10ms} + \alpha^R_{10yr} + 2\alpha^W_{10ms}) + \alpha^W_{10yr} + 2\alpha^W_{10ms}$$

The energy gain is $\frac{E_0 - E_1}{E_0} \simeq 31\,\%$.

The above simple example motivates the potential energy gain from the design compromise enabled by NVM technologies with variable retention time. This feature can be leveraged through multi-bank memory systems. Mapping variables to the most appropriate memory bank will result in significant reduction of overall memory energy. However the mapping must guarantee that the lifetimes of data stored in memory will not exceed the retention time of their allocated memory bank, in any circumstances.

1.2 Our Contribution

In this paper, we exploit static analysis and worst-case execution time techniques to estimate an upper bound of the lifetime of every store instruction in programs. Given, this information, we map each store to the most appropriate memory bank according to the most suitable NVM retention time available. Our results show that significant energy reduction can be obtained with only a few banks, favoring energy-efficient memory designs.

We summarize our contribution as follows:

- an extension of program WCET analysis with the possibility to compute the partial worst-case execution time (δ-WCET[1]) of any portion of a program;

[1]We previously described further useful applications of δ-WCETs [4]. Please note that the acronym pWCET, used previously [4], has been modified into δ-WCET in the current paper upon a suggestion from anonymous reviewers in order to avoid a confusion with "probabilistic" WCET.

- exploitation of the δ-WCET analysis to compute the worst-case lifetime for variables defined in store instructions of a program;
- multi-bank NVM memory allocation of variables based on their worst-case lifetime characterizations and memory data retention times;
- validation of the proposed approach on the Mälardalen [6] benchmark-suite (composed of 18 workloads) to show up to 80 % (and 66 % on average) dynamic energy reduction on memory, while considering design calibration parameters from NVM literature.

The rest of this paper is organized as follows. Section 2 reviews related work. We introduce necessary background on the Heptane tool in Section 3, and develop our methodology in Section 4. Finally, we evaluate our approach in Section 5 and conclude in Section 6.

2 RELATED WORK

We discuss below existing studies on the energy-efficiency of NVM-based systems and WCET estimation in general.

2.1 Energy-Efficiency of NVM Caches

There are a number of studies devoted to energy-efficiency of NVM-based caches. Reducing the impact of their expensive write operations was the main objective.

Zhou et al. in [19] proposed a technique called Early Write Termination for reducing write energy with no performance penalty. It is a write process with the capability of early termination in case of a redundant write. It is implemented at the circuit level. Smullen et al. [16] investigated an approach that focuses on technology level to redesign STT-RAM memory cells. They lower the data retention time in STT-RAM, which induces the reduction of the write current on such a memory. This enables in turn to decrease the high dynamic energy and latency of writes.

Other approaches addressed the energy-efficiency issue by considering hybrid cache memories using hardware mechanisms to migrate data between NVM and SRAM memory blocks. The idea is to keep as many write-intensive data in the SRAM blocks as possible in order to reduce the number of write operations to the NVM blocks. Li et al. [13] proposed a migration-aware compilation for STT-RAM-based hybrid cache, by re-arranging data layout to reduce the overhead of migrations. Hu et al. [8] presented an approach based on region partitioning in order to generate optimized data allocation. A program is divided into regions and before executing each region, a data allocation is generated, which is suitable for the region. In [3], we presented an approach to reduce energy consumption by eliminating so-called silent stores, i.e., store instruction instances that write to NVMs values, which were already present there. Our approach is a compile-time technique that helps to mitigate the penalty of such stores, particularly on NVMs.

2.2 WCET Estimation

WCET estimation of programs provides an upper bound of task execution time, used for guaranteeing that real-time requirements of a system are met. Traditionally, WCETs are estimated at the granularity of a function. Many tools of WCET estimation are proposed in the literature. Wilhelm et al. [18] presented an overview of methods and existing tools. Two classes of methods are distinguished: static methods and measurement-based methods. Static methods do not rely on real hardware executions. They analyze the code itself, combine the control flow graph with a model of the hardware architecture, and produce an upper bound of this combination, which is the WCET. On the other hand, measurement-based methods execute the code on real hardware or a simulator for certain inputs. Then, based on the measured times, the minimal and maximal execution times are derived.

The notion of *partial* WCET (δ-WCET) considered in the current work is recent to the field. It has been recently addressed by Jacobs et al. [10], where they focus on interference of concurrent tasks sharing a bus, and they compute for how many cycles concurrent cores may be granted access to the resource in any time interval of a given length. Avila et al. [1] also used the concept of δ-WCET, associated with the gain time. The difference between the estimated WCET and the actual execution time is known as *gain time*. Early identification of gain time requires to obtain δ-WCETs of the code instead of considering the code as a whole. The authors placed gain points in the program where they measured the actual execution time. These measurements are used to identify all sources of pessimism in the WCET analysis. Oehlert et al. [14] presented a compiler-based extraction of event arrival curves using CFGs. In order to determine the maximum/minimum number of events in a specific time interval, all possible paths in this CFG have to be considered. Their approach is an extension of the work presented by Jacobs et al. [10]. Then, they suggested an alternative idea where the objective function is set to maximize the number of events on a sub-path to be chosen in a CFG. Our work differs from this idea in that it rather aims to maximize the WCET in a given sub-path of the CFG.

3 BACKGROUND: THE HEPTANE TOOL

We consider Heptane [7], a static WCET estimation tool. The aim of Heptane is to produce upper bounds of the execution times of applications. It targets applications with hard real-time requirements (automotive, railway, aerospace domains). It computes WCETs using static analysis at the binary code

level. It is divided in two parts: HeptaneExtract and HeptaneAnalysis. HeptaneExtract generates the control flow graph G from a program compiled from C language. Then, it identifies the different loops, attaches the loop bounds information provided by the programmer and attaches the instruction addresses based on the binary file. Heptane does not include the analysis of maximum number of loops iterations, which are not always statically computable in the general case. Thus, loops must be annotated by the user with their maximum number of iterations (*maxiter*)[2]. Afterwards, HeptaneAnalysis implements IPET (Implicit Path Enumeration Technique) along with cache analysis techniques for several cache architectures [18]. Static WCET estimation methods are divided into two steps: *high-level* analysis and *low-level* analysis (see Figure 2). The *high-level* analysis consists of determining the longest execution path. The *low-level* analysis takes into consideration the micro architecture.

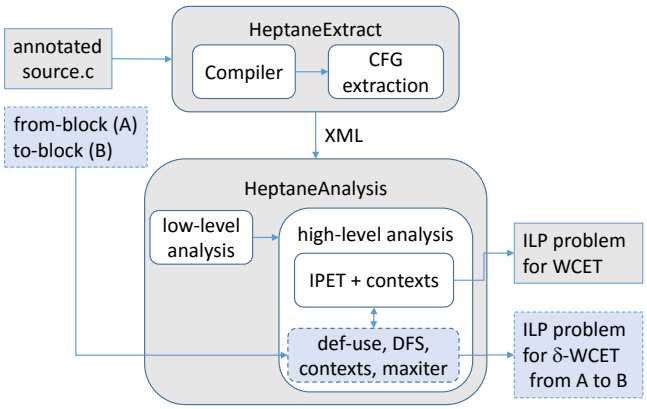


Figure 2: Heptane extended with δ-WCET analysis (blue dashed-box components added by current work).

For the *high-level* analysis, Heptane performs an IPET analysis, based on Integer Linear Programming (ILP) formulation of the WCET estimation problem. The program flow is mapped into a set of graph flow constraints. An upper bound of the program's WCET is then obtained by maximizing the following objective function: $max \sum_i n_i \times w_i$ where w_i is the timing information of the basic block i (constant in the ILP problem) determined by the low-level analysis, and n_i is the number of times the basic block i is executed (variable in the ILP problem). A basic block is a set of sequential instructions. For the *low-level* analysis, Heptane performs data address analysis, cache analysis and pipeline analysis. The pipeline analysis for all supported architectures currently considers a simple in-order pipeline. Note that Heptane performs context-sensitive analysis, which means that every call path of a function is analyzed separately, e.g. main →foo→bar, and main→bar, where bar is called directly from main but also from foo. Therefore, the objective function will be like the following: $max \sum_i n_i\_c \times w_i$, where w_i is the timing information of the basic block i in the context c and n_i_c is the number of times the basic block i is executed in the context c. The ILP problem is solved by a solver such as *lp_solve* or *cplex* by maximizing the objective function and identifying the paths that lead to the estimated WCET.

The WCET estimation depends on the CFG structure of the program, the number and type of instructions inside blocks, but also on the capability of Heptane to apply address and cache analysis to bound the number of cache misses.

[2]External tools such as oRange [5] are able to provide loops upper bounds of C programs in some cases.

4 PROPOSED METHODOLOGY

Our methodology is a two-step process, illustrated in Figure 3. First, we identify the *def-use* chains in the program (step referred to as "Reaching loads"). In other words, for each store instruction, we determine all the loads that can read the value previously written. Second, we compute the worst-case execution time between the store and all subsequent loads (step referred to as "δ-WCET"). For this purpose we have developed a method to compute partial WCET estimates which we present in Section 4.1 (further details can be found in our previous work [4]). We then present how we apply it to the case of worst-case lifetimes in Section 4.2. Both steps are entirely static analyses.

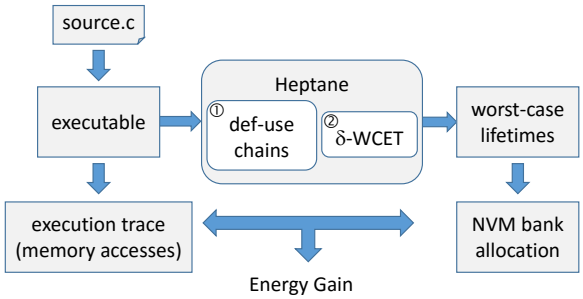


Figure 3: Sketch of our framework (input: C code).

4.1 δ-WCET Estimation

A program is given as a binary executable input (both ARM and MIPS instruction sets are supported) and an entry point (function `main`). Considering two basic blocks A and B, we are interested in estimating the WCET from block A to block B, which is the δ-WCET from A to B (item (2) of Figure 3).

The δ-WCET estimation is based on the WCET estimation. Considering a subgraph of the CFG, the objective will be to make Heptane derive the δ-WCET from its analysis for the whole CFG. Thus, we first compute the WCET estimate for the entire program, which consists in computing

Algorithm 1 General δ-WCET algorithm

procedure BetweenBlocks(A,B)
  $L = DFS(A, B)$ ▷ nodes in DFS stack at completion
  for all nodes N visited in the DFS and $N \notin L$ **do**
    $L = L \cup DFS(N, B)$
  return L ▷ the list of nodes encountered in possible paths
procedure Generation of the ILP problem from A to B
  $L = BetweenBlocks(A, B)$
  $max \sum_i n_i\_c \times w_i$ where $i \in L$ ▷ the rest of the basic blocks are excluded by setting their w_i to 0
  for all calls in L **do**
    add to the objective function the callee nodes with their callee context
  Modify w_i based on whether A and/or B are inside a loop or not ▷ Maxiter analysis

for each basic block its WCET estimate, and its worst-case execution frequency. Through this step, we obtain the system constraints of the original ILP that we want to use later for δ-WCET estimations, as well as the contextual analysis. Secondly, given two blocks A and B, we compute the set of blocks and edges that can be traversed in any path from A to B. Consider the example on Figure 1, where we selected block A=22 and B=25. All blocks 22, 23, 24, and 25 must be considered, which means that all the blocks in the different paths leading to B must be considered. We achieve this by iterating a depth-first search (DFS) on the CFG, starting from node A, until we reach B or a block that reaches B. Note that the WCET path from A to B is not necessarily on the overall WCET, i.e, the path from A to B may not be a part of the longest path in the overall program, thus, the WCET from A to B is not a sub-WCET of the program's WCET.

Then, we compose a new ILP problem for the subgraph G' obtained from the DFS, to compute the WCET from A to B (see Algorithm 1 for the details). Therefore, the objective function will include the nodes in G' along with the callee nodes (with the callee context) if there is a function call. Moreover, in order to tighten the WCET (make it less pessimistic), we analyze the *maxiter* annotations. The potential for improvement comes from the fact that the user annotation applies to the execution of the entire function, while we consider only a subgraph. We take into account the following cases, as illustrated in Figure 1:

(1) The backedge is part of the subgraph, hence the loop may execute its maximum number of iterations on a path from A to B. This is the case of the store instruction with ID 0 in Figure 1(b). We keep the value of *maxiter* unmodified.
(2) The backedge is not part of subgraph, as examplified by the store with ID=1 in Figure 1(b): the lifetime created in block 22 (initialization of variable i) is killed in block 23 by store ID=4 (i++). We set maxiter=0 so that the ILP formulation for the subgraph does not consider pessimistic frequencies due to the loop structure.
(3) The backedge is part of subgraph, but it does not contribute to any cycle. Consider for example the lifetime created by store with ID=3 (storing the result of b=b+i). Hence, we set maxiter=1.

So, the new ILP problem, presented as: $max \sum_i n_i\_c \times w_i$ where $n_i \in BetweenBlock(A, B)$, has a new system constraints, slightly different from the original one to consider the above cases. Note that the maxiter modification is applied to all backedges that are not part of the subgraph. The whole implementation of the δ-WCET algorithm has been done inside Heptane, as shown in Figure 2. Given a start node A and an end node B, the high-level analysis of Heptane performs IPET analysis along with the contextual analysis to estimate WCET and then performs the depth-first search and creates a new ILP problem using the contextual analysis related to the output of the DFS and the maxiter analysis. The contextual analysis here consists of including in the ILP objective function, the callee nodes with their callee context if there is any function call in the DFS output nodes.

4.2 Linking δ-WCET to NVM Allocation

Data retention time is another parameter one can consider when designing a "non-volatile" memory. Standard STT-RAM cells usually target several years data retention period, e.g., 10-years [16] for reliability concern. However, different designs can target shorter retention times, coming with gains in memory access latency and energy as shown in Table 1.

We envision a system with memory banks designed for various energy/retention time trade-offs. Knowing at design-time the amount of time a data is needed would let a compiler allocate data to appropriate memory banks.

Assigning a write with a certain lifetime to the appropriate memory bank will help us reduce the energy consumption by avoiding expensive write energy for writes with low lifetimes. Identifying the subsequent reads for each write is done

through a dataflow analysis called *reaching definitions* which statically determines which *definitions* may reach a given point in the code (item (2) of Figure 3).

Reaching definitions are used to compute *use-def* chains and *def-use* chains:

- *use-def* chain: consists of a use, U, of a variable, and all the *definitions*, D, of that variable that can reach that *use* without any other intervening definitions;
- *def-use* chain: consists of a *definition*, D, of a variable and all the *uses*, U, reachable from that *definition* without any other intervening *definition*.

We use the *def-use* chain where a *definition d* is a store and the reachable *uses u* are the loads. A *definition d1* reaches point *u1* if there is a path from *d1* to *u1* such that *d1* is not killed along that path. In Figure 4, *d0* is a *definition* that is never killed and its *use* is *u2*, which explains why its lifetime is the highest one. However, *d1* is killed by *d4*. The *uses* of *d1* are *u1* and *u3*; hence, after *d4*, *d1* is no longer available.

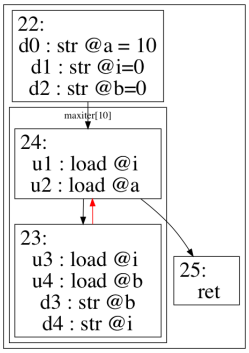


Figure 4: Example of reaching definitions

We implemented the computation of *def-use* chains in Heptane in order to compute, for every write instruction *S*, the subsequent load instructions *L1, L2..., Ln* and to define the subgraphs. Array accesses are handled in a safe (pessimistic) manner. We then create an ILP problem for every combination *S*, {*Li*} in order to estimate the δ-WCET p_i from *S* to *Li* as shown in the Algorithm 1. Then, the lifetime of the store *S* is the maximum value of p_i. We repeat this step to compute the lifetimes of all stores in the given program.

Regarding the concrete implementation of variables mapping to memory banks, both software and hardware approaches could be envisioned, as applied previously for scratchpad memories [9]. For instance, a simple solution consists in extending the instruction set with specialized load and store instructions, one for each memory bank (alternatively, the load/store could be extended with an extra parameter to specify the bank). Given the worst-case lifetimes, it is a fairly simple compiler job – or even post-processor on the assembly – to optimize each instruction. It is important that the code size does not change in order to maintain the validity of the analysis carried out by Heptane. The implementation of the above mapping solution belongs to short-term perspectives.

5 VALIDATION ON BENCHMARK-SUITE

5.1 Experimental setup

We experimented with the Mälardalen Benchmarks, a typical suite for WCET-related experiments. Benchmarks were compiled for ARM using GCC. We slightly modified the code to increase execution times which are otherwise too small: all lifetimes would be far below our lowest threshold, making gains artificially high. As customary with real-time systems, no optimization is applied (optimization level `-O0`). The reason for this is the need to keep source-level annotations consistent with the binary representation. Compiler optimizations heavily restructure the program representation, to the point that the CFG representation at binary level cannot be matched with the source level, making annotations invalid[3]. The ILP problems are solved by *cplex* and the resolution times are realistic. To assess the impact on the energy consumed by the memory, we need the number of reads and writes to all memory locations. Thus, we instrumented the benchmarks using *DynamoRIO*[4], a runtime code manipulation system that supports code transformations on any part of a program, while it executes, to obtain traces of execution. Through the traces, we count the number of occurrences for each store as well as for its subsequent loads, to estimate the energy. The overall setup is shown in Figure 3.

5.2 Architectural Setup for IoT Domain

In Internet of Things (IoT) systems, connected devices are able to sense and collect data from the environment. These devices are typically battery powered. To maintain a long period of autonomy, they must be able to manage their energy as long as possible. Therefore, the energy consumption is a critical constraint in the design of an IoT device. An IoT device is most of the time inactive, switching to low-power mode (sleep mode) and waiting for the next task. Consequently, sleep mode power will consumes an important part of energy and battery life, as the transition to the active mode will required additional energy to exit the sleep mode and become fully operational. Several Microcontrollers (MCUs) targeting low-power applications implement several power-down modes with different transition times that depend on the current low-power mode from which the MCU decides to return to the active mode. The most popular choice of NVM in an MCU is an embedded-flash memory which can

[3] Li et al. [12] traced annotations via compiler optimizations, but this requires heavy compiler machinery and is not in the focus of this work.
[4] http://www.dynamorio.org/

be used for both code-storage and data storage applications. However, as for all types of NVM, characterized by their density and low leakage, write operations are much expensive than read operations. By relaxing its retention time, we can reduce the energy cost of a write operation. Partitioning a flash memory into different blocks where each set of blocks has a retention time can help to save an important amount of the energy consumption.

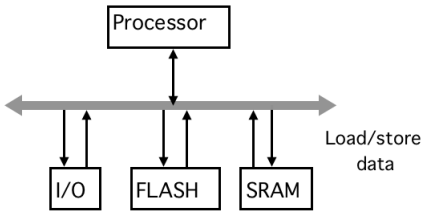


Figure 5: MCU containing a processor, SRAM, embedded Flash, and programmable I/O peripherals

In this work, we target an IoT architecture (see Figure 5), with a frequency of 40 MHz, where the read and write latency is 1 cycle and where the flash memory has a high retention time (up to 10 years). We mainly consider the memory subsystem, as in low-power designs, the core consumption of the core is extremely low. As an example, the Cortus APS25s+ core [15] is rated at 17.9 µW/MHz, i.e. 716 µW at 40 MHz. The memory systems we consider consume 2 nJ (resp. 1 nJ) per write. Assuming a write every 10 instructions at 40 MHz, the power would be $(40.10^6 \times 2.10^{-9})/10 = 8.10^{-3}W$, i.e. 8 mW (resp. 4 mW), an order of magnitude larger than the core.

5.3 Lifetimes Evaluation on Benchmarks

As a prior step to data allocation in multi-retention memory, we apply the proposed δ-WCET-based analysis to the different workloads of the Mälardalen benchmark-suite. The expected results are the lifetimes distributions according to the benchmarks. Here, by lifetimes distributions we mean the way the computed lifetimes are spread through a program. Due to the diversity of these benchmarks, the corresponding distributions in terms of store instructions vary from one benchmark to another. We can categorize the distributions according to the write-intensiveness of the benchmarks.

In this study, we distinguish a static store from a dynamic store: the former is basically an instruction, while the latter is an execution instance of the former. Figure 6 summarizes the worst-case lifetimes of static store instructions found in the subset of write-intensive workloads, i.e workloads that contain a significant number of write operations, from the Mälardalen benchmark-suite. The X and Y axes respectively denote the write instructions and their corresponding worst-case lifetime in clock cycles, while operating at a frequency of 40 MHz. We note that half of the benchmark-suite falls into this category. In order to build a few clusters of store instructions on the base of their estimated worst-case lifetimes, we consider three duration thresholds featuring three memory retention times: 26.5 µs, 3.24 s and 4.27 years, taken from Sun et al. [17] (see Table 2). It clearly appears that the identified store instructions can be partitioned into different groups w.r.t. the three lifetime thresholds. For instance, in the *fft* benchmark (top-middle), all store instruction lifetimes fall into two clusters: either below 26.5 µs threshold or below 3.24 s threshold; in the *qurt* benchmark (center), the identified lifetimes are partitioned in three clusters: either below 26.5 µs threshold or below 3.24 s threshold or above 3.24 s threshold. Therefore, this information can be leveraged to reduce energy consumption by allocating the stored data memory accesses to appropriate NVM banks.

Retention time	Read energy (nJ)	Write energy (nJ)
4.27 yr	0.085	1.916
3.24 s	0.083	0.932
26.5 µs	0.081	0.347

Table 2: 4 MB NVM memory retention times [17]

Figure 7 shows the subset of benchmarks that contain fewer static store instructions compared to those profiled in Figure 6. While the expected energy gain may a priori sound limited for write-light benchmarks, it is not necessarily true since a small number of static store instructions executed several times, e.g., in a loop, could have a non negligible impact on the overall memory energy consumption.

According to its lifetime, each store instruction will be associated with a dynamic energy cost corresponding to the memory retention threshold targeted for this instruction. From the execution traces of benchmarks, we determine how many times such instructions are executed, and we finally estimate the energy for each benchmark.

Evolution of lifetimes with program duration. As discussed in Section 1.1, not all lifetimes are equal. Some are constant, regardless of the number of iterations (such as lifetimes related to variables b and i of Figure 1(b), others vary (such as variable a). We confirmed that the same applies beyond our motivational example. Figure 8 shows how lifetimes evolve when the number of iterations of the main loop grows in *sqrt*. Clearly the first ten lifetimes remain unchanged, while the last six increase proportionally, suggesting optimization potentials, even for large run times.

5.4 Gain evaluation

To evaluate how NVMs with variable retention times have prominent impact on energy consumption, we consider different memory setups depending on the target memory bank

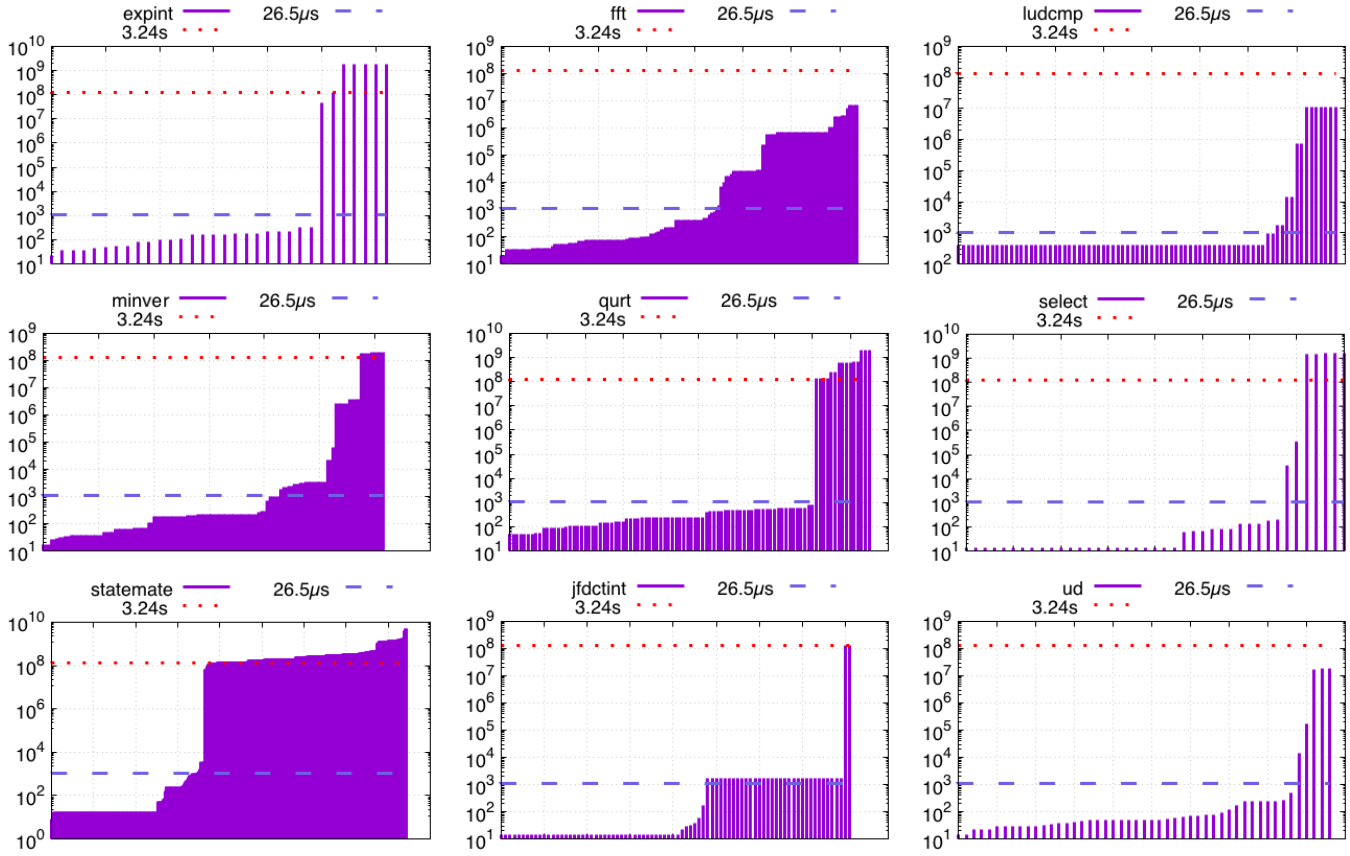


Figure 6: Worst-case lifetimes of static store instructions for write-intensive workloads from the Mälardalen benchmark-suite: the X and Y axes respectively represent the write instructions and their corresponding worst-case lifetime in cycles at 40 MHz. Two duration thresholds are made explicit along the Y axis (26.5µs and 3.24s).

Retention time	Read energy (nJ)	Write energy (nJ)
4.27 yr	0.083	0.958
3.24 s	0.032	0.466
26.5 µs	0.031	0.174

Table 3: 32 KB NVM memory retention times [17]

sizes. We already introduced one possible setup in Table 2 featuring a 4 MB STT-RAM memory size. Now, let us consider another memory setup featuring smaller memory banks with a size of 32 KB (based on the same technology [17]). The energy consumption for this new setup according to the same retention times is summarized in Table 3. Note that write energy costs overall are smaller, and read costs are also reduced for shorter retention times.

Given the above setups, we evaluate the energy gain, which mainly comes from the assignment of store operations to appropriate banks based on their worst lifetimes, hence reducing the corresponding write energy. Since the leakage of NVMs is almost null, we only focus on their dynamic energy, i.e., induced by read and write operations. We formulate the energy consumed by loads and stores as:

$$E = N^R \times \alpha^R + N^W \times \alpha^W \quad (1)$$

where N^R and N^W are respectively the number of read and write executions and α^R and α^W are respectively the dynamic energies of a read and a write operation. In a system where we have three different banks, we have three different α^R and α^W. Therefore, the energy consumed by loads and stores will become as follows:

$$E = \sum_{i \in \{\text{NVM retention times}\}} N_i^R \times \alpha_i^R + N_i^W \times \alpha_i^W \quad (2)$$

where N_i^R and N_i^W are respectively the number of read and write executions (obtained from profiling) on the NVM bank i and α_i^R and α_i^W are respectively the dynamic energies of a read and a write operation on bank i.

By applying the above formulas, we compute the dynamic energy gain, as illustrated in Figure 9 for the Mälardalen

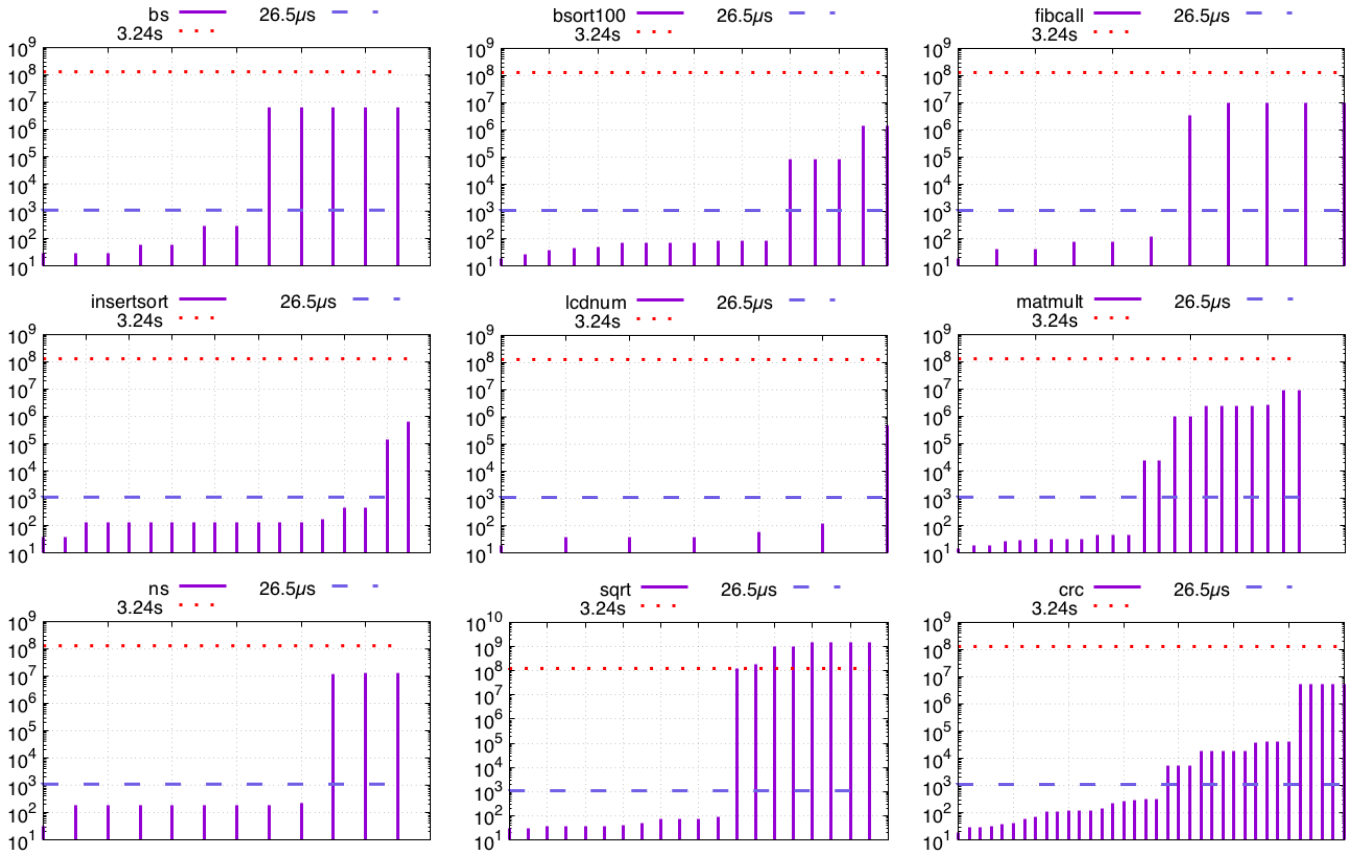


Figure 7: Worst-case lifetimes of static store instructions for write-light workloads from the Mälardalen benchmark-suite: the X and Y axes respectively represent the write occurrences and their corresponding worst-case lifetime in cycles at 40 MHz. Two duration thresholds are made explicit along the Y axis (26.5 µs and 3.24 s).

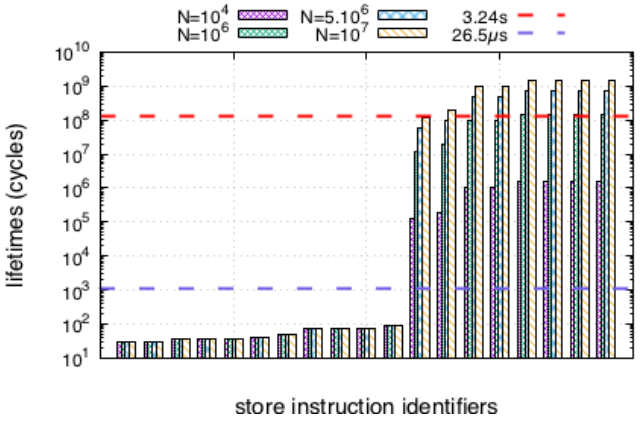


Figure 8: Lifetime distribution in *sqrt* for different values of MAXITER (i.e., N)

benchmark-suite, w.r.t. 4 MB and 32 KB STT-RAM memory setups respectively. Here, the reported gain is computed against a baseline setup consisting of an STT-RAM memory with a retention time of 4.27 years.

Figure 9 shows that we achieved with the small memory configuration up to 80 % of energy gain compared to the baseline (small memory with 4.27 years retention time) and up to 75 % with the large memory configuration. In fact, dynamic energy of read and write operations on a small size memory are less expensive compared to the large size memory.

Generally speaking, the energy gain depends on how many times a store is executed on a specific bank of memory. For illustration, Figure 10 shows for three benchmarks how many times each static store instruction has been executed (NW) and how many times the stored value has been read (NR), before getting rewritten. The trend observed for these benchmarks is representative of the overall benchmark-suite.

In the *ud* benchmark, the major part of the store instructions is executed more than 10^4 times. More than half of these instructions have low lifetimes. This is beneficial for

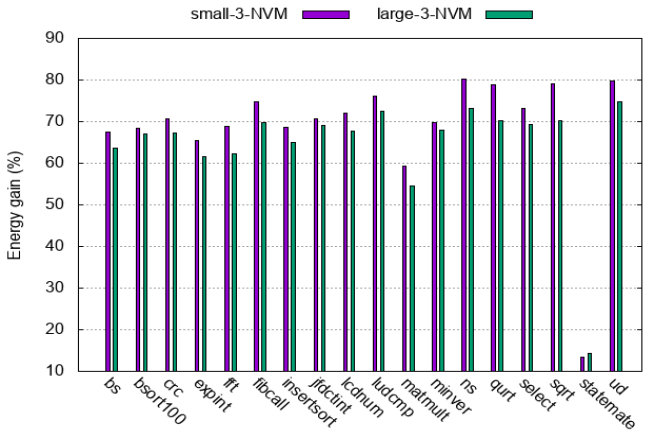


Figure 9: Energy savings based on Tables 3 and 2 setup w.r.t. a 4.27 yr STT-RAM memory

energy saving. The *statemate* benchmark, which shows more lifetimes requiring higher retention times (falling strongly in the memory region with a retention time of 4.27 years), has less energy savings among the three benchmarks. As expected, with both small and large memory configurations, the lowest energy gain is for this benchmark.

In the *matmult* benchmark, we can notice the possible detrimental impact of high NR values despite an important number of store instructions with short lifetimes. This benchmark shows a gain of 60 % as shown in Figure 9, for the small memory setup. However, this gain grows down to 55 % when considering the large memory setup, in which the cost of read energy is multiplied by more than two and half.

6 CONCLUSION AND PERSPECTIVES

In this paper, we applied a partial worst-case execution time (δ-WCET) analysis to programs in order to determine the worst-case lifetimes of program variables involved in store instructions. This information is then used to safely allocate these variables in appropriate NVM memory banks, according to their data retention time. We validated our approach on the Mälardalen benchmark-suite, by showing a significant reduction of memory dynamic energy (up to 80%, with an average of 66%). This contributes to answer the energy-efficiency challenge faced in both embedded and high-performance computing domains.

The short-term perspective to this work concerns the implementation of variables mapping to memory banks. While both software and hardware approaches could be considered, we plan to focus on the former approach. More precisely, we will explore a compiler-oriented approach to post-process the assembly for deciding the mappings based on the pre-evaluated worst-case lifetimes. Further research directions

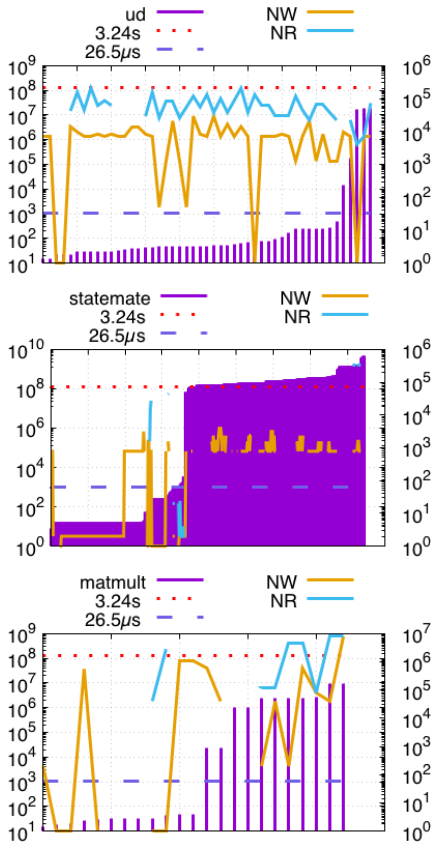


Figure 10: Number of executed static store instruction (NW) and how many times the stored value has been read (NR), before being rewritten. X, Y-left and Y-right axes are respectively store identifiers, lifetimes in cycles and the values of NR and NW.

address cache-based architectures. Indeed, the case study presented in this paper, features a cache-less system. More generally, NVMs can be exploited at different levels of the memory hierarchy. As discussed in Section 2, they can be leveraged through hybrid designs where they are combined with traditional memory technologies, e.g., SRAM or DRAM. Data layout guided by worst-case lifetimes could be key to drive each piece of data to the appropriate memory bank.

ACKNOWLEDGEMENTS

This work has been funded by the French ANR agency under the grant ANR-15-CE25-0007-01, within the framework of the CONTINUUM project.

REFERENCES

[1] Mathieu Avila, Maxime Glaizot, and Isabelle Puaut. 2013. Impact of Automatic Gain Time Identification on Tree-Based Static WCET Analysis. In *WCET*.

[2] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. 2018. Emerging NVM: A Survey on Architectural Integration and Research Challenges. *ACM Trans. Design Autom. Electr. Syst.* 23, 2 (2018), 14:1–14:32. https://doi.org/10.1145/3131848

[3] Rabab Bouziane, Erven Rohou, and Abdoulaye Gamatié. 2018. Compile-Time Silent-Store Elimination for Energy Efficiency: an Analytic Evaluation for Non-Volatile Cache Memory. In *Proceedings of the RAPIDO 2018 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Manchester, UK, January 22-24, 2018*. 5:1–5:8. https://doi.org/10.1145/3180665.3180666

[4] Rabab Bouziane, Erven Rohou, and Abdoulaye Gamatié. 2018. Partial Worst-Case Execution Time Analysis. In *ComPAS 2018 - Conférence d'informatique en Parallélisme, Architecture et Système*. Toulouse, France, 1–8. https://hal.inria.fr/hal-01803006

[5] M. de Michiel, A. Bonenfant, C. Ballabriga, and H. Cassé. 2010. Partial Flow Analysis with oRange. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (LNCS)*. 479–482.

[6] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. , 136-146 pages.

[7] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. 2017. The Heptane Static Worst-Case Execution Time Estimation Tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET)*, Vol. 8. Dubrovnik, Croatia, 12. https://doi.org/10.4230/OASIcs.WCET.2017.8

[8] Jingtong Hu, Chun Jason Xue, Qingfeng Zhuge, Wei-Che Tseng, and Edwin Hsing-Mean Sha. 2013. Data Allocation Optimization for Hybrid Scratch Pad Memory With SRAM and Nonvolatile Memory. *Trans. VLSI Syst.* (2013).

[9] Maha Idrissi Aouad and Olivier Zendra. 2007. A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy. In *2nd ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*, Olivier Zendra, Eric Jul, and Michael Cebulla (Eds.). 31–38. https://hal.inria.fr/inria-00170210

[10] Michael Jacobs, Sebastian Hahn, and Sebastian Hack. 2015. WCET Analysis for Multi-core Processors with Shared Buses and Event-driven Bus Arbitration. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*. ACM, 193–202. https://doi.org/10.1145/2834848.2834872

[11] Navid Khoshavi, Xunchao Chen, Jun Wang, and Ronald F. DeMara. 2016. Read-Tuned STT-RAM and eDRAM Cache Hierarchies for Throughput and Energy Enhancement. *CoRR* abs/1607.08086 (2016). arXiv:1607.08086 http://arxiv.org/abs/1607.08086

[12] Hanbing Li, Isabelle Puaut, and Erven Rohou. 2014. Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. In *RTNS - 22nd International Conference on Real-Time Networks and Systems*. Versailles, France. https://doi.org/10.1145/2659787.2659805

[13] Qingan Li, Jianhua Li, Liang Shi, Chun Jason Xue, and Yanxiang He. 2012. MAC: Migration-aware Compilation for STT-RAM Based Hybrid Cache in Embedded Systems. In *Int. Symp. on Low Power Electronics and Design (ISLPED)*.

[14] Dominic Oehlert, Selma Saidi, and Heiko Falk. 2018. Compiler-based Extraction of Event Arrival Functions for Real-Time Systems Analysis. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*. 4:1–4:22. https://doi.org/10.4230/LIPIcs.ECRTS.2018.4

[15] Cortus SAS. 2017. APS25s+ – Enhanced Performance Embedded Microcontroller With Leading Code Density. Product flyer – Online http://www.cortus.com/index.php/ip/.

[16] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. 2011. Relaxing Non-volatility for Fast and Energy-efficient STT-RAM Caches. In *Int. Symp. on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 12.

[17] Sun, Zhenyu et al. 2011. Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme. In *International Symposium on Microarchitecture (MICRO-44)*. ACM, 329–338. https://doi.org/10.1145/2155620.2155659

[18] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages. https://doi.org/10.1145/1347375.1347389

[19] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. Energy reduction for STT-RAM using early write termination. In *International Conference on Computer-Aided Design (ICCAD)*.